第二章 感知机（perceptron）

感知机是二分类的线形分类模型。其输入为实例的特征向量，输出为实例的类别，取+1或-1二值。感知机对应于输入空间（特征空间）中将实例划分为正负两类的超平面，属于判别类型

2.1 感知机模型

由输入空间到输出空间的如下函数

$$f(x) = sign(w * x + b)$$

w和b为感知机的模型参数

2.2 数据集的线性可分

如果有y_i = +1的实例i，$wx_i + b > 0$,对所有y_i = -1的实例i，$wx_i + b < 0$,这样的数据集T称为线形可分。而使用感知机的前提就是数据集必须是线性可分的【注意这里的x_i和y_i并不是坐标，而是特征与目标】

2.3 感知机的损失函数

所谓的学习策略就是如何来确定超平面的参数w和b。需要确定一个学习策略，即定义损失函数并将损失函数极小化。(这种思路不仅是感知机的学习策略，其实很多模型就是这种思想)

> 损失函数一个很自然的想法是误分类的总数。但是这样的损失函数不是参数w、b的连续函数，不易优化。损失函数的另一个选择是误分类点到超平面s的总距离,这是感知机所采用的。

输入空间中任意一点x_0到超平面S的距离为

$$\frac{|w*x_0+b|}{||w||}$$

如果误分类点集合M，那么所有误分类点到超平面S的总距离为

$$-\frac{\sum_{x_i\in M} y_i*(w*x_i+b)}{||w||}$$

不考虑$\frac{1}{||w||}$就得到了感知机学习的损失函数，其定义如下：

$$L(w,b)=-\sum_{x_i\in M} y_i*(w*x_i+b)$$

显然损失函数$L(w,b)$是非负的。如果没有误分类的点，损失函数值为0。而且误分类点越少，误分类点离超平面的总距离越小，损失函数值越小。

2.5 随机梯度下降法

- 假设误分类点集合M是固定的，那么损失函数$L(w,b)$的梯度由【分别对w和b求偏导】

$$\bigtriangledown_w L(w,b)=-\sum_{x_i\in M} y_i x_i$$

$$\bigtriangledown_b L(w,b)=-\sum_{x_i\in M} y_i$$

- 随机选取一个误分类点(x_i, y_i),对w, b进行更新：

$$w \longleftarrow w + \eta y_i x_i$$

$$b \longleftarrow b + \eta y_i$$

式中，η（$0 \leq \eta \leq 1$）是步长，在统计学习中又称为学习率。这个通过迭代可以期待损失函数$L(w, b)$不断减小，直到为0。

2.4 感知机学习算法

感知机学习算法是基于随机梯度下降法的对损失函数的最优化算法，有原始形式和对数形式。算法简单且易于实现。在原始形式中，首先任意选取一个超平面，然后用梯度下降法不断极小化目标函数。在这个过程中一次随机选取一个误分类点使其梯度下降。 原始算法如下：

> 输入：
>
> > 1、数据集T 2、学习率η

> 输出：w,b
>
> $$f(x) = sign(w * x + b)$$

1. 选取初值w_0,b_0
2. 在训练数据集中选取数据(x_i, y_i)
3. 如果$y_i(w * x_i + b) \leq 0$,

$$w \longleftarrow w + \eta y_i x_i$$

$$b \longleftarrow b + \eta b$$

4. 转至（2），直到训练数据集中没有误分类的点

这种算法的直观解释：当一个实例点被误分类，即位于分离超平面的错误一侧，即调整w,b的值，使得分离超平面向该误分类点的一侧移动，以减少该误分类点与超平面的距离，直至超平面越过该误分类点使其被正确分类。同时还有感知机的对偶算法（P44）

> 同时，当数据集线性可分的时候，感知机学习算法是收敛的（否则迭代结果会震荡）。但是其存在无穷多个解，这是由于多个不同的初值或不同的迭代顺序而可能有所不同。

2.5 python针对上诉算法的实现

github地址：点击跳转

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 全局变量
real_weights = 1
real_bias = 3


def generating_label_data(weight, bias,
size=10) -> pd.DataFrame:
    """
    产生需要的数据
    数据集必须是线形可分的，而不是随意的数据，具体的流
程如下：
    利用高斯白噪声生成基于某个直线附近的若干点：
x2=wx1+b
```

```
    :return:df
    """
    x1_point = np.linspace(-1, 1, size)[:,
np.newaxis]
    noise = np.random.normal(0, 0.5,
x1_point.shape)
    x2_point = weight * x1_point + bias +
noise
    input_arr = np.hstack((x1_point,
x2_point))
    # np.sign(x) 就是符号函数
    label = np.sign(input_arr[:, 1] -
(input_arr[:, 0] * real_weights +
real_bias)).reshape((size, 1))
    label_data = np.hstack((input_arr, label))
    # 转换为dataFrame
    df = pd.DataFrame(label_data, columns=
["x1", "x2", "y"])
    return df


def split_data(data, ratio) -> (pd.DataFrame,
pd.DataFrame):
    """
    数据分割
    :param data: 生成的原始整体数据
    :param ratio: 测试数据的比例
    :return:
            train_data指训练数据集
            test_data指测试数据集合
    """
    test_size = int(len(data) * ratio)
    test_data = data.loc[0:test_size, ]
    train_data = data.loc[test_size:, ]
```

```
    return train_data, test_data


def original_perceptron(x1_train, x2_train,
y_train, x1_test, x2_test, y_test, learn_rate,
train_num):
    """感知机原始算法流程如下所示
    输入：训练数据集、学习率
    输出：感知机模型
    1、选取模型的初始值
    2、在训练数据集中选取数据
    3、计算损失函数，如果小于0，则按照指定的策略对参数
模型参数进行更新，直到针对所有点的计算损失函数都大于0
    """
    # 初始化w, b
    weight = np.random.rand(2, 1)
    bias = 0
    for rounds in range(train_num):
        for i in range(len(x1_train)):
            # 算法核心：参数的迭代逻辑[这个地方的标
注的y和坐标的x和y一定要分开]
            if y_train.loc[i] * (weight[0] *
x1_train[i] + weight[1] * x2_train[i] + bias)
<= 0:
                weight[0] = weight[0] +
learn_rate * y_train[i] * x1_train[i]
                weight[1] = weight[1] +
learn_rate * y_train[i] * x2_train[i]
                bias = bias + learn_rate *
y_train[i]
        if rounds % 10 == 0:
            learn_rate *= 0.9

compute_accuracy_callback_f1(x1_test, x2_test,
```

```
y_test, weight, bias)
    return weight, bias


def compute_accuracy_callback_f1(x1_test,
x2_test, y_test, weight, bias):
    """
    计算精度：选择精度、召回率、F1
    :return:
    """
    tp = 0
    fn = 0
    fp = 0
    tn = 0
    for i in range(len(x1_test)):
        if y_test[i] != np.sign(x1_test[i] *
weight[0] + x2_test[i] * weight[1] + bias):
            if y_test[i] > 0:
                fn += 1
            else:
                fp += 1
        else:
            if y_test[i] > 0:
                tp += 1
            else:
                tn += 1
    if tp + fp == 0 & tp + fn == 0:
        return
    accuracy = tp / (tp + fp)
    callback = tp / (tp + fn)
    f1 = 2 * tp / (2 * tp + fp + fn)
    print("accuracy={0}\t\t\tcallback=
{1}\t\t\tf1={2}".format(round(accuracy, 5),
round(callback, 5), round(f1, 5)))
```

```
def data_factory(data):
    """
    返回训练与预测的数据集合
    :param data:原始整体数据集合
    :return:
    """
    train_data, test_data = split_data(data,
0.3)
    x1_train =
train_data["x1"].reset_index(drop=True)
    x2_train =
train_data["x2"].reset_index(drop=True)
    y_train =
train_data["y"].reset_index(drop=True)
    x1_test =
test_data["x1"].reset_index(drop=True)
    x2_test =
test_data["x2"].reset_index(drop=True)
    y_test =
test_data["y"].reset_index(drop=True)
    return x1_train, x2_train, y_train,
x1_test, x2_test, y_test


def main():
    size = 50  # 生成的总的数据集个数
    learn_rate = 1  # 学习率
    train_num = 100  # 训练次数
    # 生成数据
    data = generating_label_data(real_weights,
real_bias, size)
    # 获取数据
```

```
    x1_train, x2_train, y_train, x1_test,
x2_test, y_test = data_factory(data)
    # 调用模型
    original_perceptron(x1_train, x2_train,
y_train, x1_test, x2_test, y_test, learn_rate,
train_num)
    # 作图
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    for i in range(len(x1_train)):
        if y_train.loc[i] == 1:
            ax.scatter(x1_train[i],
x2_train[i], color='r')
        else:
            ax.scatter(x1_train[i],
x2_train[i], color='b')
    x = np.linspace(-1, 1.5, 10)
    y1 = real_weights * x + real_bias
    ax.plot(x, y1, color='g')
    plt.show()


if __name__ == '__main__':
    main()
```